Hashing and Hash Tables

Hashing and hash tables

- Hashing is a technique for performing `insert`, `delete`, and `find` in constant time ($O(1)$)
- A *hash table* is an array of fixed-size containing the items, which are typically key:value pairs
- Example: hash table of size $n = 10$ containing four key:value pairs
- A *hash function* is applied to the key (example: “john”), generating an index into the table, from which we retrieve the value we want (example: 25000)

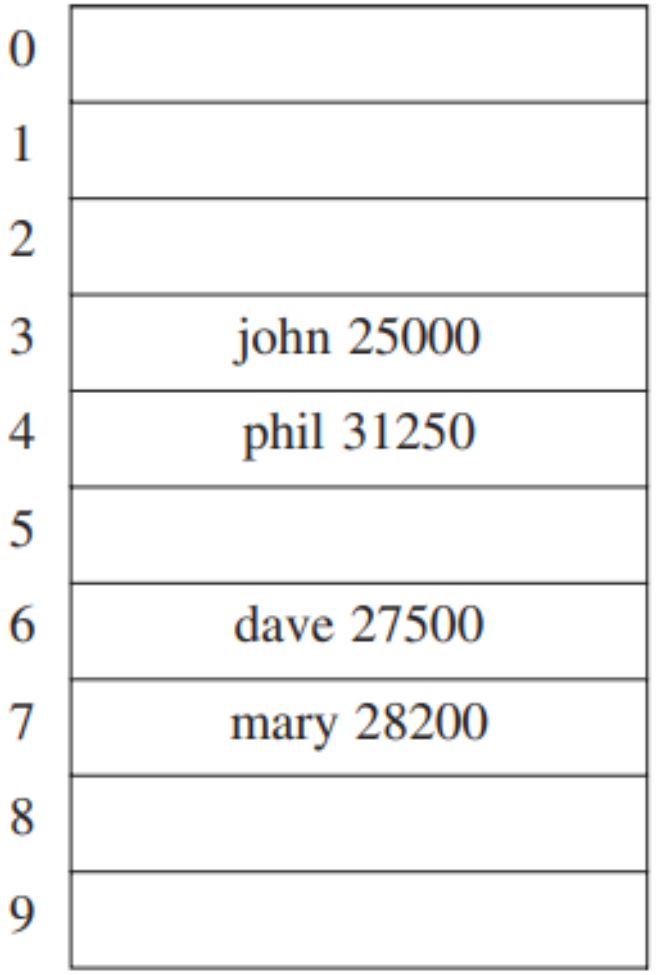


Figure 5.1 An ideal hash table

Hash function general idea:

Hash function takes an arbitrary object (could be string, int, or some other data type), converts it into a *hash code* which is an int between –inf and inf, and then compresses it down into an index between 0 and N-1, where N is the size of the hash table

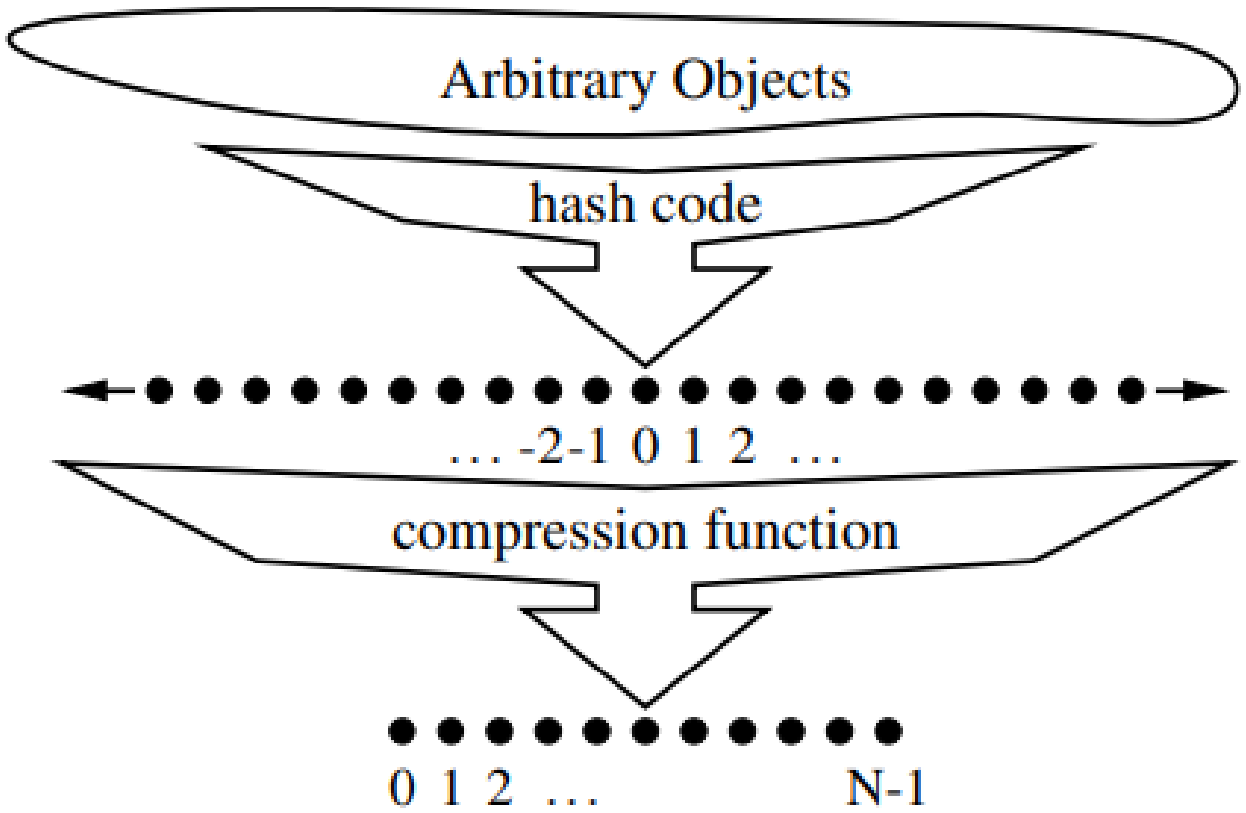


Figure 10.5: Two parts of a hash function: a hash code and a compression function.

Dec	Hx	Oct	Html	Chr
96	60	140	`	`
97	61	141	a	a
98	62	142	b	b
99	63	143	c	c
100	64	144	d	d
101	65	145	e	e
102	66	146	f	f
103	67	147	g	g
104	68	150	h	h
105	69	151	i	i
106	6A	152	j	j
107	6B	153	k	k
108	6C	154	l	l
109	6D	155	m	m
110	6E	156	n	n
111	6F	157	o	o
112	70	160	p	p
113	71	161	q	q
114	72	162	r	r
115	73	163	s	s
116	74	164	t	t
117	75	165	u	u
118	76	166	v	v
119	77	167	w	w
120	78	170	x	x
121	79	171	y	y
122	7A	172	z	z

Ascii table showing values for some chars

https://en.cppreference.com/w/cpp/utility/hash

Hash function

- If input keys are integers, can get the index into the table using key%tableSize
- Problems arise with this simple approach:
  - Example: tableSize = 10, and the keys all end in 0: key1=50,key2=60, etc.
    - 50%10 = 0
    - 60%10 = 0
    - Etc.
    - All keys hash to the same index! (bad)
- Usually, keys are strings and we need to carefully design our hash function

Figure 5.2: simple hash function for string keys. Computes answer quickly. However, if table is large, function does not distribute keys well over the table. Suppose TableSize = 10,007 (a prime number) and suppose all keys are 8 or less characters long. Since the max integer value for an ascii character is 127, this hash function can only output values between 0 and 127*8 = 1016.

```
1  int hash( const string & key, int tableSize )
2  {
3      int hashVal = 0;
4
5      for( char ch : key )
6          hashVal += ch;
7
8      return hashVal % tableSize;
9  }
```

Figure 5.2 A simple hash function

Figure 5.3: another hash function that assumes key has at least 3 characters. 27 is number of characters in English alphabet (including blank) and 729 is 27^2. The function examines only the first 3 letters, but, if these are random and the tableSize is 10,007 as before, can expect a reasonably equitable distribution. Unfortunately, English is not random and while there are 26^3 = 17576 possible combinations of three characters (ignoring blanks) the English dictionary shows only 2851 different 3-letter combinations. Even if none of these combinations collide, only 28% of the table can actually be hashed to.

```
1  int hash( const string & key, int tableSize )
2  {
3      return ( key[ 0 ] + 27 * key[ 1 ] + 729 * key[ 2 ] ) % tableSize;
4  }
```

Figure 5.3 Another possible hash function—not too good

Hash function

- Figure 5.4 shows a good hash function.
- Uses all characters in key
- Can be expected to distribute indices well over the table
- Computes

$$\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$$

- Computes a polynomial function (of 37) by use of *Horner's rule*.
  - Example: can compute

$h_k = k_0 + 37k_1 + 37^2k_2$ by formula

$h_k = \big((k_2) * 37 + k_1\big) * 37 + k_0$

```
1   /**
2    * A hash routine for string objects.
3    */
4   unsigned int hash( const string & key, int tableSize )
5   {
6       unsigned int hashVal = 0;
7
8       for( char ch : key )
9           hashVal = 37 * hashVal + ch;
10
11      return hashVal % tableSize;
12  }
```

Figure 5.4 A good hash function

tableSize

Why is it best to use a **prime number** as modulus in the hash function?

https://cs.stackexchange.com/questions/11029/why-is-it-best-to-use-a-prime-number-as-a-mod-in-a-hashing-function

Consider the set of keys $K = \{0, 1, \ldots, 100\}$ and a hash table where the number of buckets is $m = 12$. Since 3 is a factor of 12, the keys that are multiples of 3 will be hashed to buckets that are multiples of 3:

- Keys $\{0, 12, 24, 36, \ldots\}$ will be hashed to bucket 0.
- Keys $\{3, 15, 27, 39, \ldots\}$ will be hashed to bucket 3.
- Keys $\{6, 18, 30, 42, \ldots\}$ will be hashed to bucket 6.
- Keys $\{9, 21, 33, 45, \ldots\}$ will be hashed to bucket 9.

If K is uniformly distributed (i.e., every key in K is equally likely to occur), then the choice of m is not so critical. But, what happens if K is not uniformly distributed? Imagine that the keys that are most likely to occur are the multiples of 3. In this case, all of the buckets that are not multiples of 3 will be empty with high probability (which is really bad in terms of hash table performance).

This situation is more common that it may seem. Imagine, for instance, that you are keeping track of objects based on where they are stored in memory. If your computer's word size is four bytes, then you will be hashing keys that are multiples of 4. Needless to say that choosing m to be a multiple of 4 would be a terrible choice: you would have $3m/4$ buckets completely empty, and all of your keys colliding in the remaining $m/4$ buckets.

In general:

> Every key in K that shares a common factor with the number of buckets m will be hashed to a bucket that is a multiple of this factor.

Therefore, to minimize collisions, it is important to reduce the number of common factors between m and the elements of K. How can this be achieved? By choosing m to be a number that has very few factors: a **prime number**.

Collision resolution: separate chaining

- Even with a good hash function and prime tableSize, collisions will still occur (hash function will compute same index for two different keys)
- *Separate chaining* is a collision-resolution strategy that uses a linked list to store elements that hash to the same index in the table
  - If a key hashes to a currently-occupied index in the table, we append it to the linked list at that position
  - If we are searching for an key, we has the key and then traverse the linked list at the position we hashed to until we find the key we are looking for
- Example in 5.5: tableSize=10, hash function = key%tableSize
  - Insert(0),insert(81),insert(64), insert(1), etc.

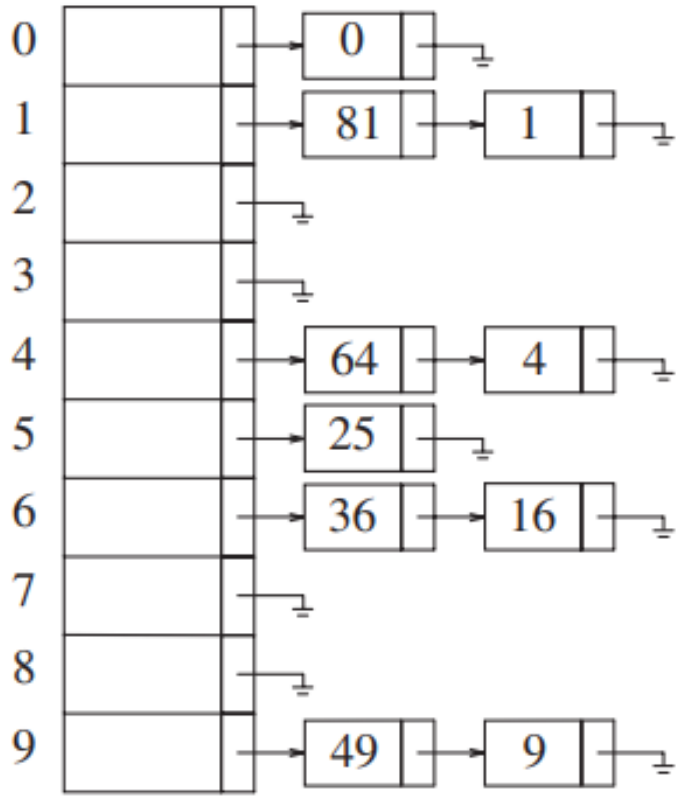


Figure 5.5 A separate chaining hash table

Collision resolution: separate chaining

- Separate chaining HashTable implementation stores an *array of linked lists*, allocated in the constructor

```
1   template <typename HashedObj>
2   class HashTable
3   {
4     public:
5       explicit HashTable( int size = 101 );
6
7       bool contains( const HashedObj & x ) const;
8
9       void makeEmpty( );
10      bool insert( const HashedObj & x );
11      bool insert( HashedObj && x );
12      bool remove( const HashedObj & x );
13
14    private:
15      vector<list<HashedObj>> theLists;     // The array of Lists
16      int currentSize;
17
18      void rehash( );
19      size_t myhash( const HashedObj & x ) const;
20  };
```

Figure 5.6 Type declaration for separate chaining hash table

Separate chaining: makeEmpty, contains, remove

```
1       void makeEmpty( )
2       {
3           for( auto & thisList : theLists )
4               thisList.clear( );
5       }
6
7       bool contains( const HashedObj & x ) const
8       {
9           auto & whichList = theLists[ myhash( x ) ];
10          return find( begin( whichList ), end( whichList ), x ) != end( whichList );
11      }
12
13      bool remove( const HashedObj & x )
14      {
15          auto & whichList = theLists[ myhash( x ) ];
16          auto itr = find( begin( whichList ), end( whichList ), x );
17
18          if( itr == end( whichList ) )
19              return false;
20
21          whichList.erase( itr );
22          --currentSize;
23          return true;
24      }
```

Separate chaining: insert

```
1       bool insert( const HashedObj & x )
2       {
3           auto & whichList = theLists[ myhash( x ) ];
4           if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
5               return false;
6           whichList.push_back( x );
7
8               // Rehash; see Section 5.5
9           if( ++currentSize > theLists.size( ) )
10              rehash( );
11
12          return true;
13      }
```

Figure 5.10 `insert` routine for separate chaining hash table

Hash tables without linked lists

- Separate chaining has the disadvantage that it uses linked lists
- Could slow the algorithm due to time needed to allocate nodes and requires implementation of a separate data structure
- An alternative to separate chaining: if collision occurs, try alternative cells until an empty spot is found
- Formally:
- Cells $h_0(x), h_1(x), h_2(x)$, … are tried in succession
- $h_i(x) = \big(hash(x) + f(i)\big)\ \%\ \text{tableSize}$
- Function f is the collision resolution strategy

Linear probing

- In linear probing, f is a linear function of i, typically $f(i) = i$
- Try cells sequentially (with wraparound at end of table) in search of an empty cell
- Example: inserting keys {89,18,49,58,69} into hash table with tableSize=10 and hash function=key%tableSize

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Hash table with linear probing, after each insertion

Linear probing performance

- As long as table is big enough, a free cell will always be found
- *Primary clustering* is a problem that occurs with linear probing. The table becomes full in certain regions, and it takes longer and longer to find an empty spot to put the element (Example: indices 8,9,0,1,2 from previous slide
- Performance degrades as *load factor* λ increases
- Load factor λ = number of elements in table divided by tableSize
  - Example: tableSize=10, number of elements in table=9, $\Rightarrow \lambda = 9/10$
- Expected number of probes using linear probing is $\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$ for insertions and unsuccessful searches, and $\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$ for successful searches
- if $\lambda = 0.9$, we need 50 probes on average to insert 1 element
- If $\lambda = 0.75$, we need 8.5 probes on average to insert 1 element

Linear probing performance

- Performance of linear probing (dashed line) for unsuccessful (U) and successful (S) searches vs random probing
- Random probing not a real strategy (just generates random indices after a collision until an empty spot is found), but serves to illustrate how linear probing results in primary clustering, increasing the number of expected probes
- λ should remain below 0.5 to ensure good performance

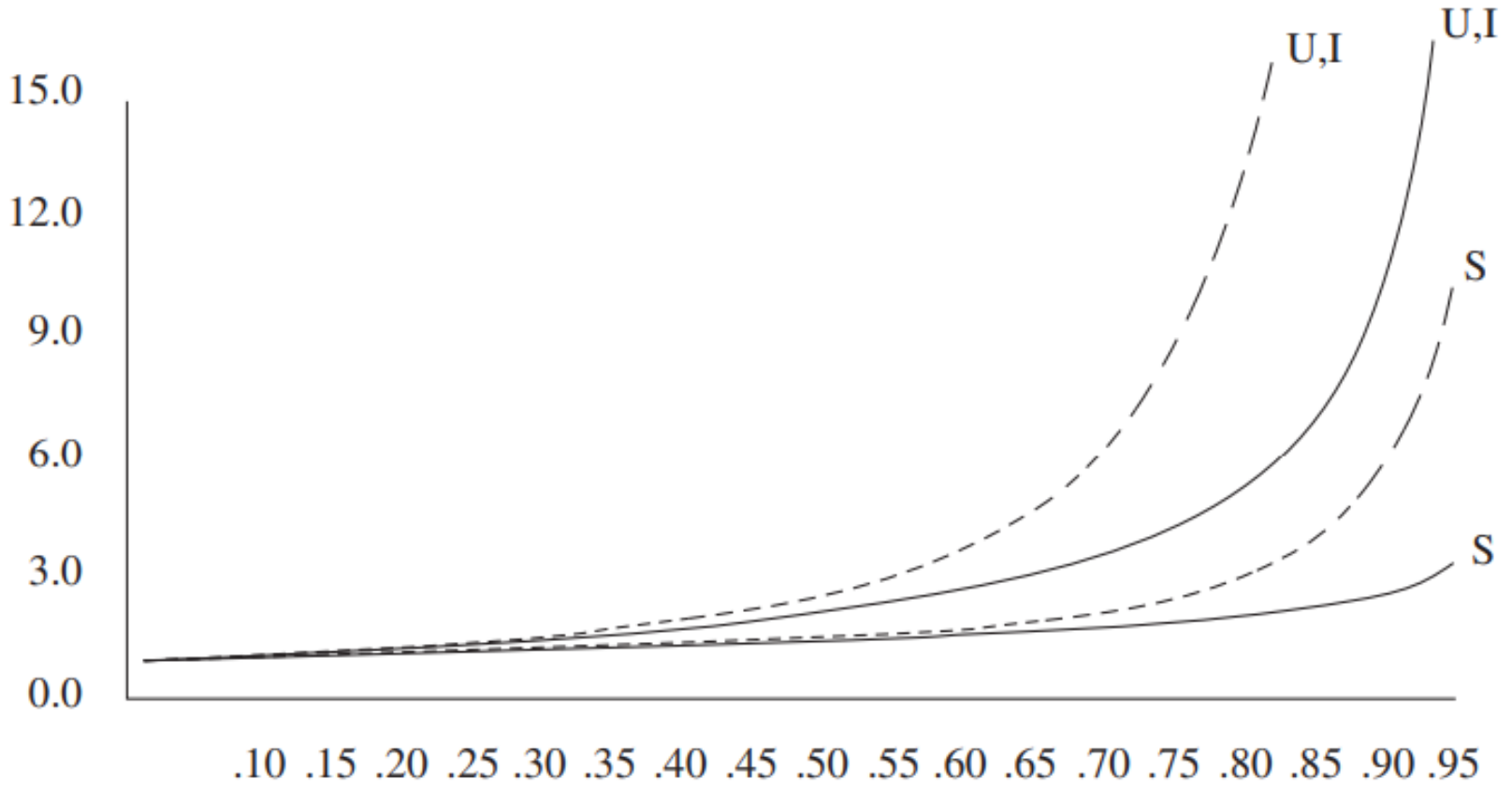


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

Quadratic probing

- Quadratic probing: a collision resolution method that eliminates primary clustering problem of linear probing
- Collision function is quadratic: $f(i) = i^2$
- λ cannot exceed 0.5 for quadratic probing! Otherwise, it is possible that insert will fail (might never find an empty cell)
    - Page 204 of Weiss textbook shows proof that an empty cell can always be found if tableSize is prime and $\lambda \leq 0.5$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.13 Hash table with quadratic probing, after each insertion

Quadratic probing: table initialization and `contains`

```
1       explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
2         { makeEmpty( ); }
3
4       void makeEmpty( )
5       {
6           currentSize = 0;
7           for( auto & entry : array )
8               entry.info = EMPTY;
9       }
```

Figure 5.15 Routines to initialize quadratic probing hash table

```
1       bool contains( const HashedObj & x ) const
2         { return isActive( findPos( x ) ); }
3
4       int findPos( const HashedObj & x ) const
5       {
6           int offset = 1;
7           int currentPos = myhash( x );
8
9           while( array[ currentPos ].info != EMPTY &&
10                 array[ currentPos ].element != x )
11          {
12              currentPos += offset;  // Compute ith probe
13              offset += 2;
14              if( currentPos >= array.size( ) )
15                  currentPos -= array.size( );
16          }
17
18          return currentPos;
19      }
20
21      bool isActive( int currentPos ) const
22        { return array[ currentPos ].info == ACTIVE; }
```

Figure 5.16 `contains` routine (and `private` helpers) for hashing with quadratic probing

Quadratic probing: `insert` and `remove`

```
1       bool insert( const HashedObj & x )
2       {
3               // Insert x as active
4           int currentPos = findPos( x );
5           if( isActive( currentPos ) )
6               return false;
7
8           array[ currentPos ].element = x;
9           array[ currentPos ].info = ACTIVE;
10
11              // Rehash; see Section 5.5
12          if( ++currentSize > array.size( ) / 2 )
13              rehash( );
14
15          return true;
16      }
```

```
18      bool remove( const HashedObj & x )
19      {
20          int currentPos = findPos( x );
21          if( !isActive( currentPos ) )
22              return false;
23
24          array[ currentPos ].info = DELETED;
25          return true;
26      }
```

Double hashing

- Another collision resolution method is *double hashing*
- $f(i) = i \cdot hash_2(x)$
- Apply a second hash function ($hash_2$) to key, and probe at distance $hash_2(x)$, $2 \cdot hash_2(x)$, etc.
- A common choice for $hash_2$ is $R - (x \,\%\, R)$ where R is a prime smaller than tableSize

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.18 Hash table with double hashing, after each insertion

R=7 in this example. First collision occurs when 49 is inserted. $hash_2(49) = 7 - 0 = 7$ so 49 is inserted in position 6. $hash_2(58) = 7 - 2 = 5$ so 58 is inserted at position 3. Finally, 69 collides and is inserted at distance $hash_2(69) = 7 - 6 = 1$ away.

Rehashing

- If table gets too full, running time for operations will start taking too long, and insertions could even fail if we're using quadratic probing
- Solution: expand the table by 2x and re-insert all elements using a new hash function
- Figure 5.19-20:
  hash function = key % 7
- Figure 5.21:
  hash function = key % 17
- This operation is called *rehashing,* and is $O(n)$

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 5.19 Hash table with linear probing with input 13, 15, 6, 24

0	6
1	15
2	23
3	24
4	
5	
6	13

Figure 5.20 Hash table with linear probing after 23 is inserted

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 5.21 Hash table after rehashing

Hashing in the STL (unordered_set and unordered_map)

Unordered Map

Unordered maps are associative containers that store elements formed by the combination of a ***key value*** and a ***mapped value***, and which allows for fast retrieval of individual elements based on their keys.

In an `unordered_map`, the ***key value*** is generally used to uniquely identify the element, while the ***mapped value*** is an object with the content associated to this ***key***. Types of ***key*** and ***mapped*** value may differ.

Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their ***key*** or ***mapped*** values, but organized into ***buckets*** depending on their hash values to allow for fast access to individual elements directly by their ***key values*** (with a constant average time complexity on average).

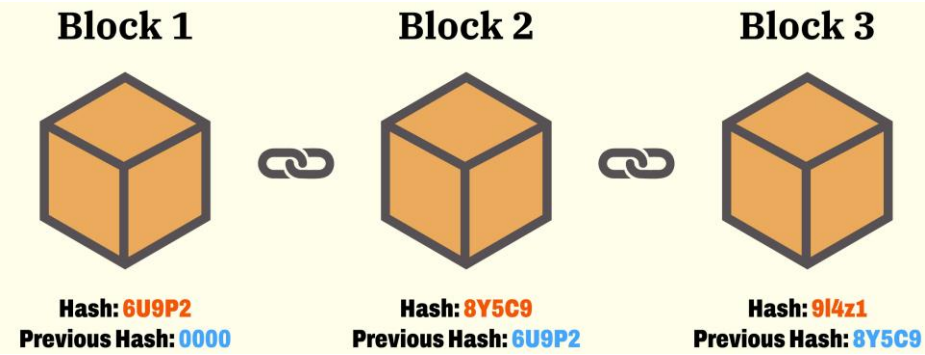
`unordered_map` containers are faster than `map` containers to access individual elements by their ***key***, although they are generally less efficient for range iteration through a subset of their elements.

Unordered maps implement the direct access operator (`operator[]`) which allows for direct access of the ***mapped value*** using its ***key value*** as argument.

`Hash`

A unary function object type that takes an object of type ***key type*** as argument and returns a unique value of type size_t based on it. This can either be a class implementing a ***function call operator*** or a pointer to a function (see constructor for an example). This defaults to hash<Key>, which returns a hash value with a probability of collision approaching `1.0/std::numeric_limits<size_t>::max()`.
The `unordered_map` object uses the hash values returned by this function to organize its elements internally, speeding up the process of locating individual elements.
Aliased as member type `unordered_map::hasher`.

Bitcoin mining



- "mining bitcoin" consists of repeatedly computing a hash function on a block of data until the result of the hash function satisfies some requirement
- Example of SHA outputs:

```
SHA224("The quick brown fox jumps over the lazy dog")
0x 730e109bd7a8a32b1cb9d9a09aa2325d2430587ddbc0c38bad911525
SHA224("The quick brown fox jumps over the lazy dog.")
0x 619cba8e8e05826e9b8c519c0a5c68f4fb653e8a3d8aa04bb2c8cd4c
```

- To successfully mine a bitcoin, find the 'nonce' value that, when combined with pre-existing block, yields an output starting with n zeros (for example, if $n = 5$, output must be 0x00000…
  - "difficulty level" of mining depends on n, higher values of n = more difficult to mine